

# Supporting Model-Driven Refactoring

Michaël Hoste

Service de Génie Logiciel, Université de Mons  
6, Avenue du Champ de Mars, B-7000 Mons BELGIUM  
Email: michael.hoste@umons.ac.be

**Abstract**—The purpose of Model-Driven Software Engineering is to work on models rather than on code. This approach allows developers to raise the level of abstraction and to hide the accidental complexity of the code. Our work explores the ways to help users improve model quality by introducing refactorings during the evolution of models while keeping them consistent. The tool we extend is EMF Refactor, an Eclipse plug-in designed to create refactorings visually and to apply them on Eclipse EMF Models.

## I. INTRODUCTION

Some years ago, Fowler introduced the concept of refactoring to improve object-oriented code. He defined refactoring as “the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure” [1]. Nowadays, refactoring is included out-of-the-box in a whole range of Integrated Development Environments like IntelliJ IDEA, Eclipse and Netbeans. Those IDE’s can apply pre-defined refactorings to, among other things, make the code less complex, improve its reusability and its readability.

The current trend is to abstract the code and to use models as primary entities during the development of software. Doing so allows designers to cope with the inherent complexity of software systems while raising the level of abstraction and hiding the accidental complexity as much as possible [2].

The number of tools that allow us to work directly on models has been increasing every day. However, the creation and the use of refactorings is still not significantly developed. Users should be able to apply existing refactorings on models as well as to create new ones with little previous experience. In practice, however, this is not really the case.

The main goal of our research is to create, or to adapt, a framework that could assist users during model-driven software evolution. This framework will help them to choose the right refactoring at the right time and to apply it at the right place. It will also ensure that a refactoring keeps the model consistent while improving its quality.

## II. TOOLS

Since we are only at the early stages of our research, we are still looking for a good candidate framework to adapt. We decided to use the Eclipse Modeling Framework (EMF) for our first experiments because it provides a set of useful tools to deal with models. On top of EMF, we use two more

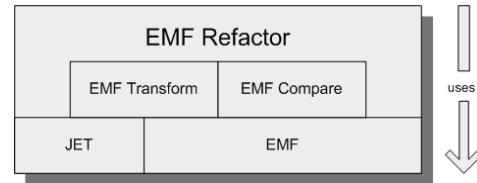


Fig. 1. Architecture of Eclipse plug-ins for model transformation

plug-ins for Eclipse : *EMF Transform*<sup>1</sup> and *EMF Refactor*<sup>2</sup>. The first one is being developed by the *Technische Universität Berlin* and its purpose is to apply in-place EMF model transformations based on the graph transformation formalism [3], [4], [5]. Model transformations are visually specified and are quite easy to learn. The second one, created at the *Philipps-Universität Marburg*, uses EMF Transform and acts at a higher level of abstraction to provide pre-defined refactorings on EMF Models. The architecture of these plug-ins that operate on top of Eclipse is depicted in Figure 1.

We intend to use these plug-ins to create new integrated refactorings for other types of UML diagrams such as state machine diagrams and component diagrams. We have to ensure that these new refactorings do not introduce inconsistencies in the other diagrams that compose the model. In the future, our objective is also to provide an interactive environment that will assist users in the task of selecting appropriate model refactorings with the help of *model metrics* and *model smells*. Model metrics, which are the equivalent of software metrics, are numerical values used to assess some quality aspects of models. Model smells, which are the equivalent of bad smells, are “structures in the code<sup>3</sup> that suggest (sometimes scream for) the possibility of refactoring” [1].

## III. REFACTORING DEFINITION

Before considering the possibility of applying a refactoring with EMF Refactor, it is necessary to define the corresponding model transformation with EMF Transform. Since a model refactoring is a special kind of model transformation (one that preserves the behaviour), the designer needs to make sure this transformation preserves the behaviour of the model (cf. Section IV-C).

Model transformations in EMF Transform are defined using transformation rules. Each transformation rule is divided into

<sup>1</sup><http://user.cs.tu-berlin.de/~emfrans/>

<sup>2</sup><http://www.mathematik.uni-marburg.de/~swt/modref/>

<sup>3</sup>In our case : *model*

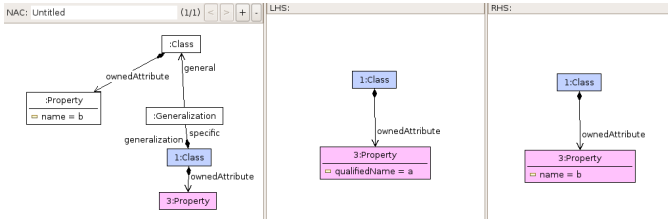


Fig. 2. Rename Attribute transformation with EMF Transform

three parts : *Left-Hand Side* (LHS), *Right-Hand Side* (RHS) and *Negative Application Conditions* (NAC). LHS corresponds to the model transformation preconditions that need to be satisfied for the application of the rule. RHS corresponds to the model transformation postconditions, i.e., the resulting part of the transformation. NAC's, which are the opposite of LHS, are conditions that prevent the rule from being applied. In other words, if one of the NAC's is satisfied, the rule cannot be used.

Figure 2 graphically depicts how to define a simple *rename attribute* transformation in a UML class diagram with EMF Transform. The NAC ensures that a parent class does not already have an attribute with the new name. This transformation can then be deployed as a new integrated refactoring in the EMF environment with EMF Refactor.

#### IV. REFACTORING HANDLING

Refactoring handling is a difficult activity that includes various tasks. Firstly, it must detect which part of the model is the worst in terms of quality and then find out which refactoring is best suited to remove this lack of quality. Then, after the refactoring is applied, the behaviour and consistency of the model need to be checked.

The framework we intend to build will automatise all these activities so users will not need to care about them. It will detect model smells and use them to find parts of the model with poor quality. Then, it will propose many refactoring scenarios so that users will only have to choose which one they want to apply.

The next sections briefly describe each of the activities that must be handled by our framework.

##### A. Where to apply a refactoring

In order to know where it is useful to apply a refactoring on a model, we need to be aware of its qualities and weaknesses. Evaluating the quality of a model proves to be a delicate task. This requires to define some quality criteria according to the objectives : performance, readability, modularity, ... Sometimes, the quality criteria are in contradiction with each other and cannot be jointly satisfied. To detect this contradiction in criteria, we can use the theory of critical pair analysis to find different model smells that cannot be simultaneously resolved by applying refactorings [6].

To avoid this issue, we need to prioritise quality criteria. For example, in a real-time application, the performance criterion

will have higher priority than the readability criterion and the refactorings proposed to the user will be adapted accordingly.

##### B. Which refactoring to apply

The choice of a model refactoring to apply depends on the quality criterion we want to improve on a model fragment. For example, if a model smell indicates a *data class*, i.e., a class with significantly more data than behaviour (more attributes than methods), the candidate refactorings will be the ones that are going to relocate foreign methods to this data class.

As well as finding the best suited refactoring to improve a model, it may be necessary to evaluate the model after its refactoring to see if its global quality did not decline. Indeed, many new model smells can be introduced when resolving another one and the resulting quality can become worse. If the model quality is worse after a refactoring, it could be wise to undo it and try to find a better way to refactor the model.

##### C. Behaviour preservation

By definition, a refactoring needs to preserve the behaviour of the model. Unfortunately, the notion of model behaviour is not well-defined in literature. Also, it depends on the modeling context : real-time software has temporal constraints, embedded software has hardware constraints and critical software has safety constraints. These constraints need to be taken into account when checking the behaviour.

There are formal techniques to check the behaviour of a model with logic programming [7], logic predicates [8] or constraint satisfaction problems (CSP) [9], [10].

Another more practical way to check the behaviour preservation of a model is to transform it into an entity that can be tested like source code. You can check the behaviour of the source code by using unit tests. If your unit tests cover all your code and are still good after refactoring, then you can assume the behaviour of the code (and of the linked model) is still the same [11], [12]. This method can work when a model is used to generate code but is not appropriate for other situations.

##### D. Consistency preservation

In general, a model is build from different views. UML, among others, uses this multi-view representation. Each model view is represented by one UML diagram. If a refactoring is applied to a diagram, it is likely that it will no longer be consistent with other diagrams of the same model. To avoid this issue, either the refactoring must be defined to deal with multi-view representations or the inconsistencies must be corrected after the refactoring application.

#### V. CONCLUSION

This article introduced some tools developed on top of the Eclipse Modeling Framework, a popular framework in the models community. Our objective is to contribute to the development of these tools by introducing new refactorings for state and component diagrams. We will also improve the framework so it could assist the user during model evolution and refactoring handling. The main idea is to propose the best

refactoring scenarios to the user with the help of model metrics and model smells. Those scenarios will represent the best ways to improve model quality.

Our goal is to adapt the framework so it could support multi-view refactorings while preserving the behaviour and consistency of a model.

Our main objective with this project is to automatically find a refactoring scenario that will fix the highest number of model smells to get the best resulting model quality while dealing with quality criteria contradictions that can be introduced.

#### ACKNOWLEDGEMENTS

I would like to thank Pr. Tom Mens for all the pieces of advice he gave me about how to write articles in English.

Many thanks to Gabriele Taentzer and her researchers from the University of Marburg for the warm welcome at their University and for sharing their knowledge.

Funded by the Actions de Recherche Concertées – Ministère de la Communauté française - Direction générale de l'Enseignement non obligatoire et de la Recherche scientifique.

#### REFERENCES

- [1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.
- [2] F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering*, 20th ed. Addison-Wesley, 1995.
- [3] S. Jurack and G. Taentzer, "ROOTS: An Eclipse plug-in for graph transformation systems based on AGG," in *Proc. Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, 2007, pp. 491–496.
- [4] T. Mens, "On the use of graph transformations for model refactoring," in *Generative and transformational techniques in software engineering (GTTSE)*, ser. Lecture Notes in Computer Science, R. Lämmel, J. Saraiva, and J. Visser, Eds., vol. 4143. Springer-Verlag, 2006, pp. 219–257.
- [5] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, and E. Weiss, "EMF Model Refactoring based on Graph Transformation Concepts," 2006.
- [6] T. Mens, G. Taentzer, and O. Runge, "Detecting structural refactoring conflicts using critical pair analysis," in *Preproceedings of the Workshop on*, p. 105.
- [7] M. Proietti and A. Pettorossi, "Semantics preserving transformation rules for prolog," in *Proc. ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation (PEPM)*. ACM, 1991, pp. 274–284.
- [8] A. Pretschner and W. Prenninger, "Computing refactorings of state machines," *Software and Systems Modeling*, vol. 6, no. 4, pp. 381–399, 2007.
- [9] M. Van Kempen, M. Chaudron, D. Koudrie, and A. Boake, "Towards proving preservation of behaviour of refactoring of UML models," in *Proc. SAICSIT*, 2005, pp. 111–118.
- [10] G. Engels, R. Heckel, J. Küster, and L. Groenewegen, "Consistency-Preserving Model Evolution through Transformations," in *Proc. Int'l Conf. Unified Modeling Language (UML)*, J.-M. Jézéquel, H. Hussmann, and S. Cook, Eds. Springer-Verlag, 2002, pp. 212–227.
- [11] A. van Deursen and L. Moonen, "The video store revisited – thoughts on refactoring and testing," in *Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering (XP)*, 2002, pp. 71–76.
- [12] J. U. Pipka, "Refactoring in a "test first"-world," in *Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering (XP)*, 2002.